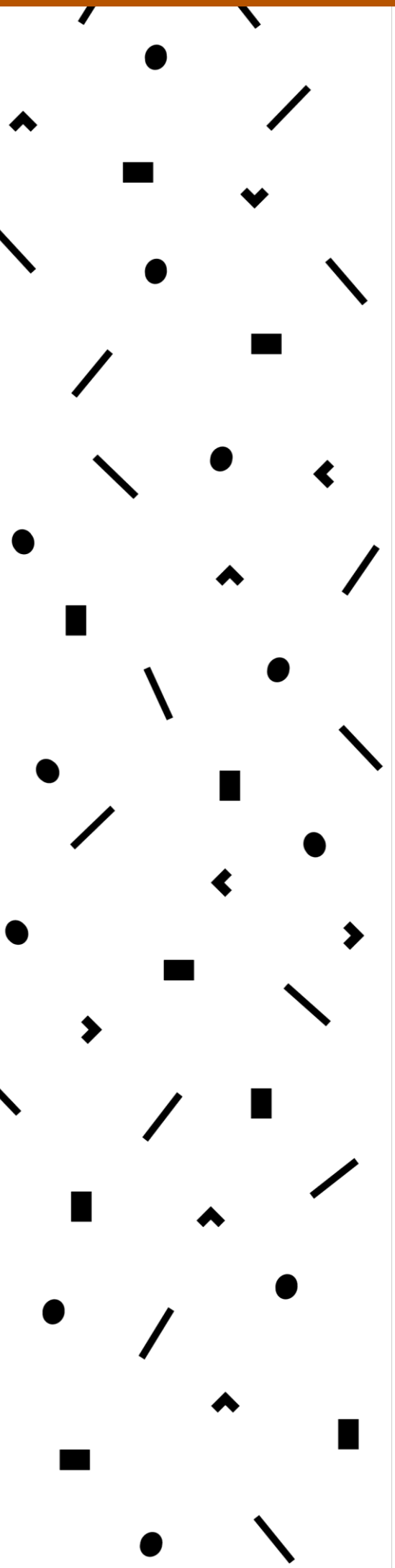


LAMK Lahden ammattikorkeakoulu
Lahti University of Applied Sciences

WEBSHOP ENGINE

Design and implementation of an eCommerce platform front office

LAHTI UNIVERSITY OF APPLIED SCIENCES
Faculty of Technology
Degree Programme in Information Technology
Software Engineering
Bachelor's thesis
Spring 2016
Hans Selenius

Lahti University of Applied Sciences
Degree Programme in Information Technology

SELENIUS, HANS:

Webshop engine
Design and implementation of an eCommerce platform front office

Bachelor's Thesis in software engineering, 52 pages

Spring 2016

ABSTRACT

This Bachelor's thesis was commissioned by DevNet Oy and deals with the company's eCommerce suite. The thesis focuses on the design and implementation of a webshop, or more specifically, a webshop front office as part of the company's larger eCommerce suite.

The eCommerce suite consists of three parts: the webshop itself, a back office integrated into the company's ERP system, and a REST API for database transactions and communication between the two. The webshop was designed using well-known, modern frameworks with the Laravel MVC framework at its core for the back end, together with the AngularJS framework for the front end logic. The shop is easily maintained, changed and expanded with new features and functionality by a software developer. It also features changeable themes in order to be easily customized for the company's various end users.

At the time of writing, the webshop application is undergoing customization for deployment by several of the company's clients and the first implementations are expected to go online during early spring 2016.

Key words: Laravel, AngularJS, MVC, eCommerce

Lahden ammattikorkeakoulu
Tietotekniikan koulutusohjelma

SELENIUS, HANS:

Verkkokauppasovellus
Verkkokaupan tekninen sunnittelu ja toteutus

Ohjelmistotekniikan opinnäytetyö, 52 sivua

Kevät 2016

TIIVISTELMÄ

Opinnäytetyön toimeksiantajana toimi DevNet Oy. Opinnäytetyö käsittelee yhtiön verkkokauppa-alustaa keskittyen itse verkkokauppasovelluksen tekniseen suunitteluul ja toteutukseen.

Verkkokauppa-alusta on jaettu kolmeen osaan: varsinainen verkkokauppasovellus, firman ERP-järjestelmä verkkokaupan ohjauspaneeliina, sekä REST API -alustan tietokantakyselyjä ja edellä mainittujen osien kommunikointia varten. Verkkokauppa on suuniteltu hyödyntäen moderneja ja tunnetuimpia ohjelmistoviitekehyksiä, ja sovelluksen ydin on rakennettu Laravel- ja AngularJS MVC -sovelluskehysten ympärille. Verkkokauppa on helposti ylläpidettävissä, räätälöitävissä ja muokattavissa uusilla ominaisuuksilla sovelluskehittäjän näkökulmasta. Sovellus tukee myös eri ulkoasuteemojen käyttöä, jotta kauppa olisi helposti räätälöitävissä firman eri asiakkaita varten.

Tällä hetkellä verkkokauppa räätälöidään useampien asiakkaiden lopullista käyttöönottoa varten. Ensimmäisten oikeiden julkaisuiden odotetaan tapahtuvan aikaisin kevään 2016 aikana.

Asiasanat: Laravel, AngularJS, MVC, eCommerce

TABLE OF CONTENTS

1 INTRODUCTION

The purpose of the project is the design and implementation of a successor to DevNet Oy's previous eShop eCommerce platform.

DevNet Oy (after this referred to as "DevNet") is an Information Technology (IT) solutions company based in Lahti, Finland. The company, established in 2005, specializes in enterprise scale software development services with a primary focus on, but not limited to, the web, i.e. websites and other web-based software solutions. The company also offers hosting services, e.g. webhotels and various server solutions, provided under DevNet's auxiliary business name, WMHost. DevNet also has a branch in Jyväskylä, Finland, since spring 2014. (DevNet Oy 2016.)

The new eCommerce suite will have its back office directly integrated into the company's existing Enterprise Resource Planning (ERP) software in order to centralize all of the end users' business management into one big system. The software suite will also feature an API (Application Programmable Interface) for RESTful (Representational State Transfer) communication with different front office clients including, but not limited to, the company's own implementations.

This study focuses on the structure and technical design as well as the frameworks used for building the eCommerce platform's front office, i.e. the webshop itself (after this referred to as "the shop").

2 OPERATIONAL ENVIRONMENT

2.1 Customer requirements

As decided by the company, the new eCommerce software suite will be consisting of 3 different parts/applications (Figure 1): a REST API for the database record transactions, the actual webshop itself (i.e. the front office, which this study focuses on), and a back office built as an extension to the company's current ERP system. The software suite in question is built on top of a LAMP (Linux, Apache, MySQL/MariaDB, PHP) stack, in accordance with current company policies.

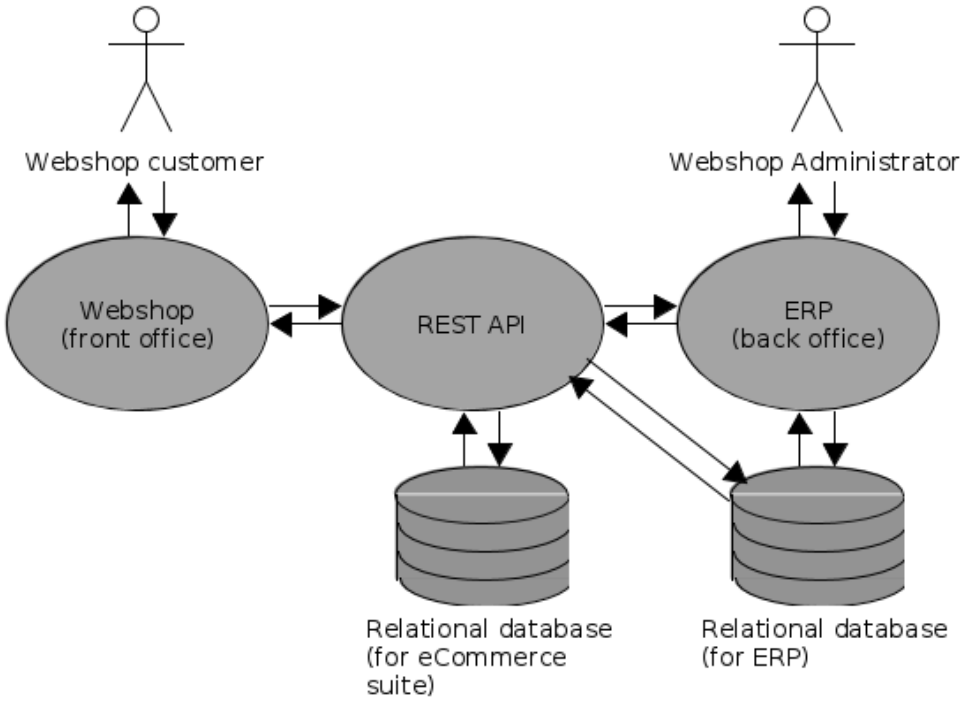


FIGURE 1. The eCommerce suite and its components

As depicted in Figure 1, both the front office and the back office will be utilizing the same centralized REST API for communication and for the eCommerce suite's database transactions. The ERP system, being more than merely a back office for the eCommerce suite, has its own database. However, apart from the stock management (in the ERP system's database, which can also be accessed through the API), all data specific

to the eCommerce suite will be stored in its own, dedicated database accessed solely through the API. Furthermore, the front office is designed using well-known, modern frameworks, which are to be centralized around the application's Laravel core (as Laravel is the main framework used for the back end and core structure of the application). The front office should also to be easily maintained, changed and/or expanded with new features and functionality from a developer's point of view and should feature changeable themes in order to be easily customized for the company's various end users.

2.2 Frameworks

A web "framework" is a type of foundation designed to help developers build and provide important core functionality common to web applications. The shop application attempts to make use of some of the more well known frameworks in order to standardize workflow and make it as accessible and maintainable as possible for both future webshop implementations and developers. This chapter shortly describes the main frameworks selected for the shop application.

2.2.1 Laravel

Laravel is an open source PHP framework created by Taylor Otwell, which incorporates many of the best features from other well-known PHP framworks such as CodeIgniter, Symphony, Zend, Yii, etc. The first version was released in 2011. (Surguy 2013.)

Despite being a relatively new framework, Laravel has in a short time become a standard-bearer for development of modern PHP applications. It utilizes the Model-View-Controller (MVC) design pattern along with the object-oriented programming paradigm and features such properties as modularity, testability and configuration management. It also comes with powerful capabilities for routing, a fluent query builder, Object Relation Mapper (ORM) and ActiveRecord implementation, built-in authentication, a template engine named Blade (partly inspired by ASP's Razor template

language) as well as a command bus, which makes it easy to dispatch events. (Bean 2015.)

The back end of the shop is built using Laravel 5 (version 5.1 at the time of writing) and handles the server size processing, HTTP requests and routing for the shop, as well as the data transactions between the shop and the eCommerce platform's REST API.

2.2.2 AngularJS

AngularJS is a popular open source JavaScript framework sponsored and maintained by Google Inc., which has been used in some of the largest and most complex web applications around (Freeman 2014).

AngularJS is an MVC framwork, where JavaScript objects are controllers, the Document Object Model (DOM) are views and the object properties store model data. It also features data binding, dependency injection and directives. (Green & Seshadri 2013.)

AngularJS (version v1.4.8 at the time of writing) is used as the main framework for the front end of the shop and most of the custom JavaScript logic makes use of it. It is used for templating, presenting and sorting of data, AJAX (Asynchronous JavaScript and XML) requests and interactivity, as well as data-bindings to HTML elements.

2.2.3 jQuery

Though most of the JavaScript used by the shop comes in the form of AngularJS, jQuery (version v.2.1.4 at the time of writing) is also available for use and is mostly used for various jQuery plugins (such as the OWL Carousel slider).

jQuery is a lightweight, cross-browser JavaScript library that simplifies processes like DOM manipulation and event handling (The jQuery Foundation 2016). Despite being depicted as a library rather than an actual "framework", it is still included in this list because of its worldwide

status and sheer usage and market share statistics. As of March 2016, jQuery holds a JavaScript library market share of 95.9%, while the absolute usage percentage across all websites is 69.4% (Q-Success 2016).

2.2.4 Bootstrap

Bootstrap is a responsive front-end framework for web applications. It contains a grid system and styling for various user interface components such as buttons, navs and tables. It also features many JavaScript plugins for adding interactive elements. (Spurlock 2013.)

Bootstrap (version v3.3.5 at the time of writing) is the main framework used for the themes, user interface and front end styling of the shop, as well as some of the JavaScript components.

2.3 Dependency and asset managers

While the shop does feature a lot of custom written code, it also makes use of various third-party assets, dependencies, modules and plugins in both its back and front ends. Therefore, since manually copying and moving all those assets around into their respective directories and files by hand would be a tedious and tiring task (also very likely more prone to errors), the shop uses a handful of various dependency and asset managers in order to automize and ease the installation and maintenance of said assets and dependencies.

2.3.1 Composer

Composer is a dependency manager for PHP. It allows installation and management of various libraries on a per-project basis. (Adermann & Boggiano.)

Composer is used to manage the PHP libraries and components used by the back end of the shop, including Laravel.

2.3.2 Node and NPM

Node, also referred to as Node.js, is a well-known server-side JavaScript environment based on Google Inc.'s runtime and is used for event-driven applications. Node also comes with NPM (Node Package Manager), its own package manager and repository for third-party dependencies and modules. (Teixeira 2012.)

The shop does not really use Node for its back end code or custom programming logic (since that is done using Laravel and PHP, as opposed to Node and JavaScript), but for its package manager, NPM. Since the shop requires access to a wide array of various modules and tools, NPM is used to provide that possibility through its many packages. For instance, Gulp and Bower (descriptions further below) both rely on Node.

2.3.3 Gulp

Gulp is a streaming build system. It is built with Node and can automate and organize various development tasks quickly. (Maynard 2015.)

Gulp is used to keep track of, as well as compile and minify the shop's many JavaScript, CSS and SASS assets, both vendor/third-party and custom, into just a few, concentrated files. The Laravel framework also provides its own API, Elixir, for defining basic Gulp tasks (Otwell 2016e).

2.3.4 Bower

While Composer handles the back end assets of the shop, the various front end assets, libraries and plugins are mainly managed by Bower.

Bower is a package manager optimized for the front end that manages components that contain HTML, CSS, JavaScript, fonts and images. Bower requires Node, NPM and Git (described further below). (Bower 2016.)

3 STRUCTURE, THEMING AND TEMPLATING

3.1 Hierarchial structure

Being a Laravel application for the most part, the shop follows the same directory and information flow structure as provided by Laravel. Below is an in-depth description of said structure and customizations for the shop.

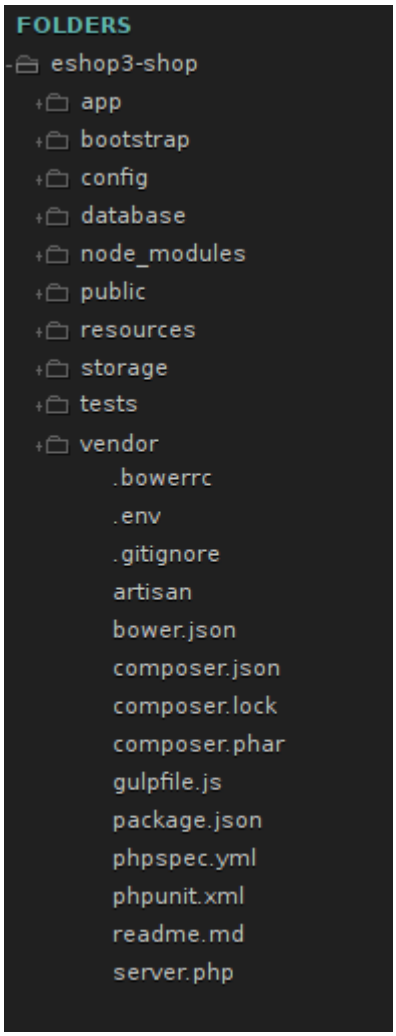


FIGURE 2. The root directory

Figure 2 shows the application's root directory structure (as provided by Laravel 5.1):

- app: Contains the core code (Otwell 2016a). Most of the custom directories and back end code for the shop is found inside this

directory and its subdirectories. In-depth description of the subdirectories and their structure can be found further below.

- bootstrap: A few files that boostrap and configure autoloading (Otwell 2016a).
- config: Contains the configuration files for the application (Otwell 2016a). This directory contains a lot of changeable custom settings for the shop, such as product display settings, file paths, order settings etc, for quick and easy configuration by the developer.
- database: Contains database migration and seeds (Otwell 2016a). Not used by the shop, since all database transactions and other tasks are handled through the separate eCommerce API.
- node_modules: Contains the various Node modules and assets for the shop.
- public: Contains the front controller and various assets, such as fonts, images, JavaScript files and stylesheets (Otwell 2016a). The compiled and minified SASS (CSS) and JavaScript, both vendor and custom, that are presented to the front end and seen by the browser are found inside this directory and its subdirectories. In-depth description of the subdirectories and structure can be found further below.
- resources: Contains the view / Blade template files, as well as the raw assets (SASS and JavaScript) and language files (Otwell 2016a). The resources directory also contains all theme specific files for the shop. This includes the theme's own SASS files, AngularJS controllers, Angular services and other JavaScript files. The themes and their structure are described more thoroughly in their own chapter.
- storage: Contains various caches, logs and compiled Blade templates (Otwell 2016a).
- tests: Contains automated tests (Otwell 2016a).
- vendor: Contains the Composer dependencies (Otwell 2016a).

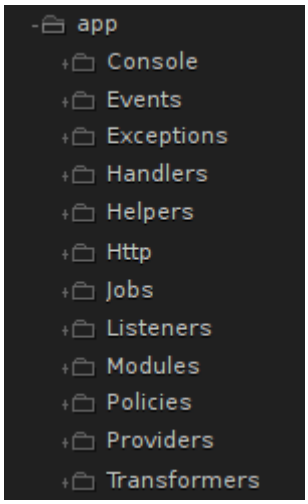


FIGURE 3. The app directory

Figure 3 shows the app directory, which holds most of the custom back end logic and contains the following subdirectories (only the first sublevel listed). The directories marked with an asterisk (*) in the detailed description below were not added by default (by Laravel):

- Console: Contains the Artisan commands (Otwell 2016a).
- Events: Contains event classes. Events can be used to tell the application that some action has occurred (Otwell 2016a). For instance, the shopping cart fires an CartWasUpdated event to notify the rest of the shop that there has been some sort of change to it that should be taken into account.
- Exceptions: Contains the exception handlers (Otwell 2016a).
- Handlers *: Contains the custom event handlers for the shop, such as the SetCartUpdatedCookie handler, which is called when the CartWasUpdated event is fired.
- Http: The controllers, the middleware and the requests (Otwell 2016a). The Models are also stored under this directory (Http/Models).
- Jobs: Queable jobs for the application (Otwell 2016a).
- Listeners: Contains the handler classes for the events (Otwell 2016a). Not currently in use for any custom code. The custom event handlers are found inside the Handlers directory instead.

- Modules *: Contains the custom back end modules (services and facades). Described more thoroughly in their own chapter.
- Policies: Not used.
- Providers: Contains the back end service providers (Otwell 2016a). The custom service providers for the shop are always part of a module and are found under Modules instead, i.e. the shop does not use this directory for its custom providers
- Transformers *: Contains the “transformer” classes for the shop. Described more thoroughly in their own chapter.

3.2 Themes

The shop’s themes are designed to be easily switched, extended and modified by a software developer. The themes for the shop consist mainly of views in the form of Blade templates, styles CSS (vendor) and SASS (custom styles) and JavaScript (mainly AngularJS for the custom code).

3.2.1 Blade templates

Blade templates have their own syntax for control structures (conditionals, loops and such), extending layouts, displaying data etc (although use of regular PHP code inside the template file is also supported). Blade also supports template inheritance and sections, meaning that it is possible to have a “main” layout file that contains and includes “sections” and/or sub-views in other files. (Otwell 2016b.)

Blade templates use the @-sign for various control structures and template features, for instance:

- @if and @else for conditional statements
- @extends to extend another template file
- @section to define a section
- @yield to display the contents of a given section
- @include to include a sub-view

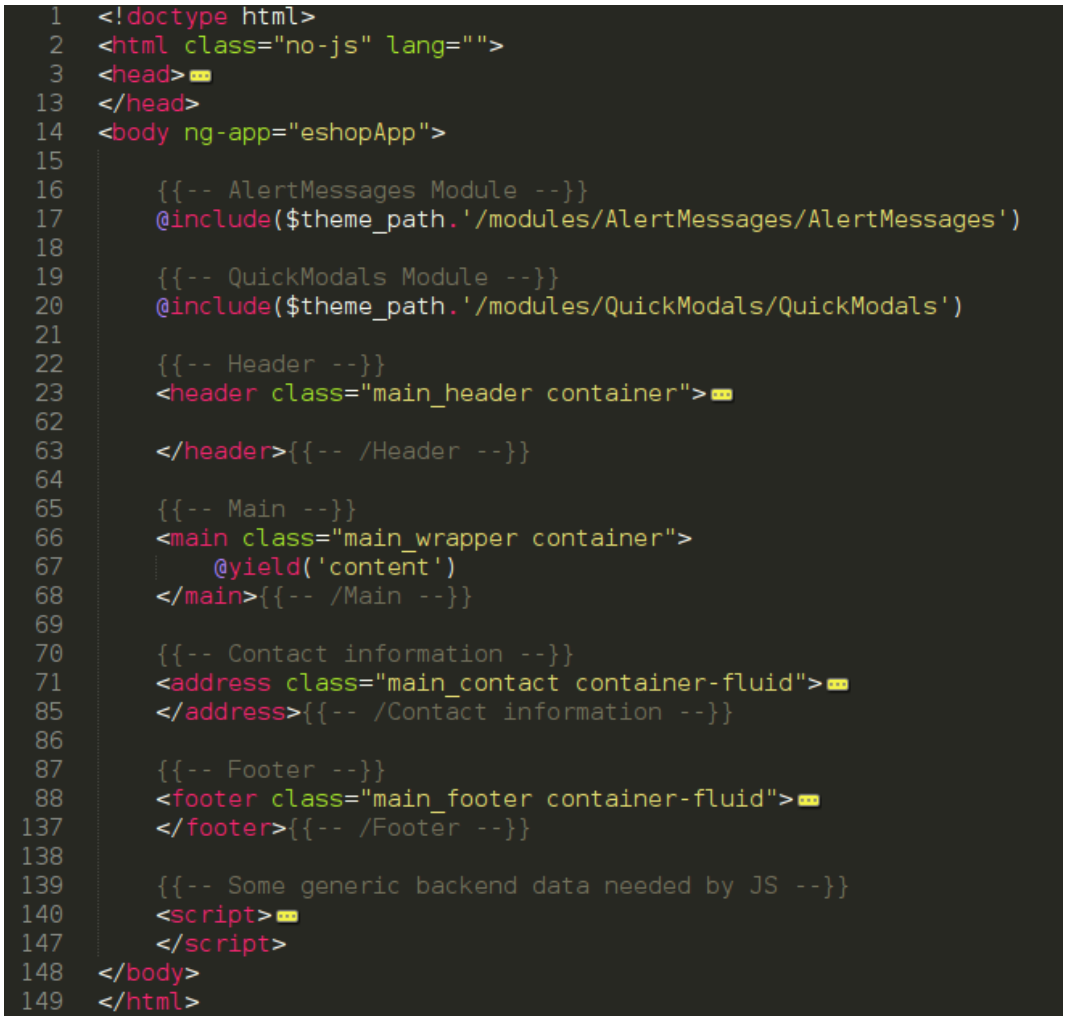
The templates use curly braces for displaying data, such as PHP variables and results from functions, for instance:

- {{ $my_variable }} to echo the contents of a variable
- {{ date('Y') }} to echo the result of PHP's date() function

Since many JavaScript frameworks also utilize the curly braces, Blade can be instructed to ignore a statement with curly braces by adding an @-symbol before the braces. (Otwell 2016b.)

Seen below is an actual example of how the shop utilizes the Blade templating engine with some takes from the source code of the shop's default template (notice that the figures only contain the rows necessary to illustrate an understandable example. This is done for clarity, since the full files contain tens or even hundreds of rows of code depending on the file. Parts where rows are missing are indicated with "…").

The main template/layout file (defalt.blade.php) shown in Figure 4 uses the @yield directive (line 67) to display the contents of the product page (Figure 5). The product page (product.blade.php) uses the @extends directive (line 1) to extend the main layout (default.blade.php) and @section to define the "content" and "title" sections (lines 6 and 10) of the main layout. It also uses the @if and @else conditionals (lines 2 and 4) to check that the product data exists and to print data from the product array using the curly braces syntax (e.g. line 23):



```
<!doctype html>
<html class="no-js" lang="">
<head>...
</head>
<body ng-app="eshopApp">

    {{-- AlertMessages Module --}}
    @include($theme_path.'/modules/AlertMessages/AlertMessages')

    {{-- QuickModals Module --}}
    @include($theme_path.'/modules/QuickModals/QuickModals')

    {{-- Header --}}
    <header class="main_header container">...

    </header>{{-- /Header --}}

    {{-- Main --}}
    <main class="main_wrapper container">
        @yield('content')
    </main>{{-- /Main --}}

    {{-- Contact information --}}
    <address class="main_contact container-fluid">...
    </address>{{-- /Contact information --}}

    {{-- Footer --}}
    <footer class="main_footer container-fluid">...
    </footer>{{-- /Footer --}}

    {{-- Some generic backend data needed by JS --}}
    <script>...
    </script>
</body>
</html>
```

FIGURE 4. The main template file

```
@extends($theme_path.'.layouts.default')
@if(!isset($product))
    <strong>THIS VIEW REQUIRES PRODUCT DATA TO BE LOADED IN
    ADVANCED OR INJECTED INTO VIEW ON LOAD</strong>
@else

    @section('title')
        {{ $product['name'] }}
    @endsection

    @section('content')
        <div class="product_page">

            <div class="row">

                <div class="col-xs-12 col-sm-3 col-md-3
                product_image">

                    {{-- ProductImageSwitcher Module --}}
                    @include($theme_path.'/modules/
                        ProductImageSwitcher/ProductImageSwitcher',
                        compact($product))
                </div>

                <!-- Product info -->
                <div class="col-xs-12 col-sm-6 col-md-6">
                    <h1>{{ $product['name'] }}</h1>
                    <article>
                        {!! $product['description_long'] !!}
                    </article>
                </div>

                @if(!$product['treat_as_category'])
                @endif

            </div>

            @if($product['treat_as_category'] || $product['parent_id
                '])
            @endif

        </div>
    @endsection

@endif
```

FIGURE 5. The product page

3.2.2 File and directory hiearachy

The files specific to the selected theme are found under resources/views/themes/<theme name> (where <theme name> is replaced with the actual name of the theme). This is shown in Figure 6.

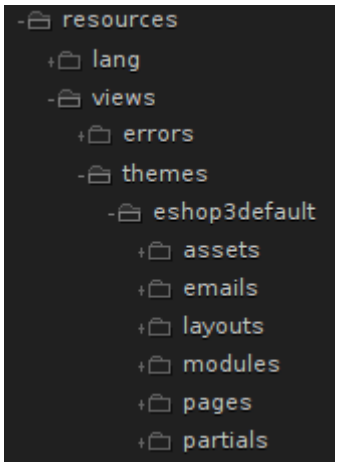


FIGURE 6. The theme specific directory

The views in this directory are divided into the following subdirectories/categories:

- emails
- layouts
- pages
- partials
- modules

The emails directory contains the HTML emails for the theme, such as the order confirmation and password reset emails.

The layouts directory contains the main template file, default.blade.php, and holds the default layout for the whole site. The pages, partials and front end modules are all included/called inside this file.

The pages directory contains specific page layouts, such as the layout for the product presentation page (product.blade.php) or the checkout page (checkout/checkout1.blade.php). Most pages are loaded as a section or part of the main template file, but they can also be stand alone pages (if the rest of the main theme is not needed for that particular page/view). The pages are named using lowercase and underscores, e.g. product.blade.php for the product information page.

The partials directory contains minor content blocks and/or sub-views such as a menu or sidebar. For instance, partials/TopNav/TopNav.blade.php contains the top menu for the shop and can be hooked into the layout using Blade's @include directive. Notice that the partial also has its own directory, since there might be different variations of it used in different places. Partials that require an extensive amount of JavaScript logic, AJAX and own styling are referred to as "modules", rather than "partials" and are not found inside the partials directory, as they have their own directory and logic. The partial files and directories are named using the Pascal case capitilzation style, e.g. TopNav, MyPartial, etc.

The modules directory contains the directories and sub-views specific to the modules. Like the partials, modules can also be included just about anywhere using the @include directive, but usually require more extensive logic than a simple partial. Modules are described more thoroughly in their own chapter. The module files and directories are named using the Pascal case capitalization style, e.g. QuickCart, MyModule, etc.

The SASS and JavaScript files for the theme are under the resources/themes/<theme name>/assets where they go into their respective js and sass subdirectories. The key styles for the theme are found inside the sass directory's theme.scss and variables.scss files.

The theme.scss file contains the base styles and look for the theme and variables.scss contains some basic SASS variable values for the theme, such as the theme's base colors. The variable file is also a good place to put any possible overrides of the Boostrap SASS variables. To avoid clogging up the main stylesheet for the theme, the sass directory is also further divided into its own subdirectories specific to modules, pages and paritials, for when there is an extensive amount of custom styles required for anything that fits under the description of those entities.

Like the sass directory, the js directory also contains its own module subdirectory for module specific scripts (AngularJS files). Furthermore, the js directory also holds the theme's angular-default.js file, where the

AngularJS app used across the whole site as well as which AngularJS third-party modules (not to be confused with the shop's own modules) to load are defined. The directory also contains a helpers.js file for some generic JavaScript helper functions.

3.2.3 Theme switching and compilation

The theme for the shop can be easily switched by changing theme settings (name and path of the theme) under config/theme.php. and in the gulpfile.js file. The theme path can be fetched from the config or through the globally shared $theme_path variable whenever a view is loaded inside a Laravel controller or template. This makes it easy for the shop to always knows in which theme's directory to look. Figure 7 shows a view being loaded in the PagesController controller (app/Http/Controllers/PagesController.php) on line 43. Figure 8 shows the TopNav partial (Blade sub-view) loaded in the main template file (default.blade.php) using the @include directive on line 45.

```
29      /**
30       * Display the home page
31       *
32       * GET /
33       *
34       * @return Response
35       */
36      public function home()
37      {
38          $page = Eshop::api_get('pages/id/' . $this->id['home']);
39          $page = json_decode($page->getContent());
40
41          $page = $this->pageTransformer->transform($page->data);
42
43          return view($this->theme_path . 'pages/home')->withPage(
                $page);
44
45      }
```

FIGURE 7. Loading a view

```
43                  <div class="col-xs-12 col-sm-12 col-md-7 col-lg-8">
44                      {{-- TopNav Partial --}}
45                      @include($theme_path.'/partials/TopNav')
46                  </div>
```

FIGURE 8. Loading a partial

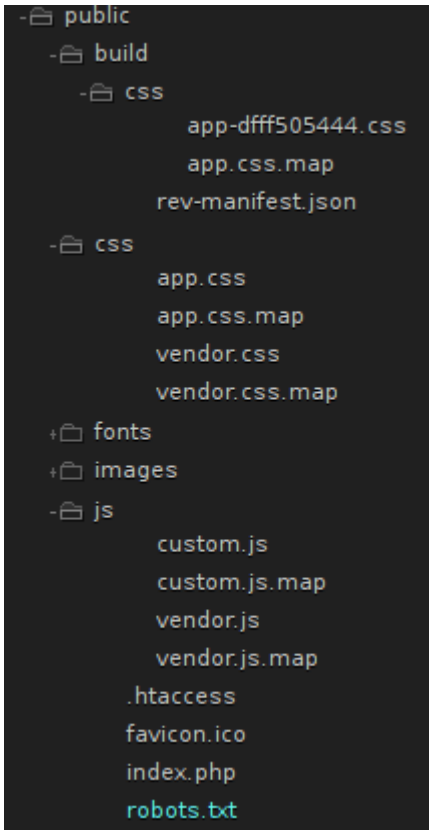


FIGURE 9. The public directory

The gulpfile (gulpfile.js) is the settings file for Gulp and Laravel Elixir. It keeps track of all the “non-Blade-template” assets (i.e. SASS, CSS, JavaScript, fonts, etc), or rather their input and output locations and names, as well as the name of the theme itself (a variable in the file that needs to be changed when the theme changes). When Gulp is run (from the command line), it compiles and minifies all JavaScript and SASS + CSS assets into the following public files (as seen in Figure 9):

- public/js/vendor.js: vendor/third-party scripts
- public/js/custom.js: deveoper’s own custom scripts
- public/css/vendor.css: vendor/third-party styles
- public/build/css/app-<unique suffix>.css: developer’s own custom styles

The <unique suffix> part of the filename is automatically replaced with a unique hash for so called “cache-busting”, i.e. to force the web browser to load a fresh file instead of a possible cached one (Otwell 2016e). The right asset can then be automatically loaded inside the

template using Laravel's global elixir function (Otwell 2016e). Figure 10 shows the assets being loaded in the default.blade.php view of the shop's theme. The stylesheets (vendor.css and app-<unique suffix>.css) and scripts (vendor.js and custom.js) are being loaded inside the <head> element. Note that Laravel's elixir helper function is being used to load the right asset for app.css.

```
8       <link rel="stylesheet" href="{{ asset('css/vendor.css') }}" />
9       <link rel="stylesheet" href="{{ elixir('css/app.css') }}" />
10
11      <script src="{{ asset('js/vendor.js')}}" defer></script>
12      <script src="{{ asset('js/custom.js')}}" defer></script>
13  </head>
```

FIGURE 10. Loading the assets

4 APPLICATION WORKFLOW AND FUNCTIONALITY

4.1 MVC

Laravel is a Model-View-Controller (MVC) architecture framework (Principe & Yoon 2015). Being a Laravel application at its core, the shop relies on MVC for its information workflow.

The MVC approach basically means separating the application components into different areas of logic, i.e. “views”, “controllers” and “models”, thereby allowing for easier maintenance and expansion. In a traditional description of MVC (Figure 11), the controller handles the business logic and input and updates the model when necessary, the model oversees the data and relays information about its changes to the controller and/or view, and the view handles the presentation of said data to the user (Principe & Yoon 2015).

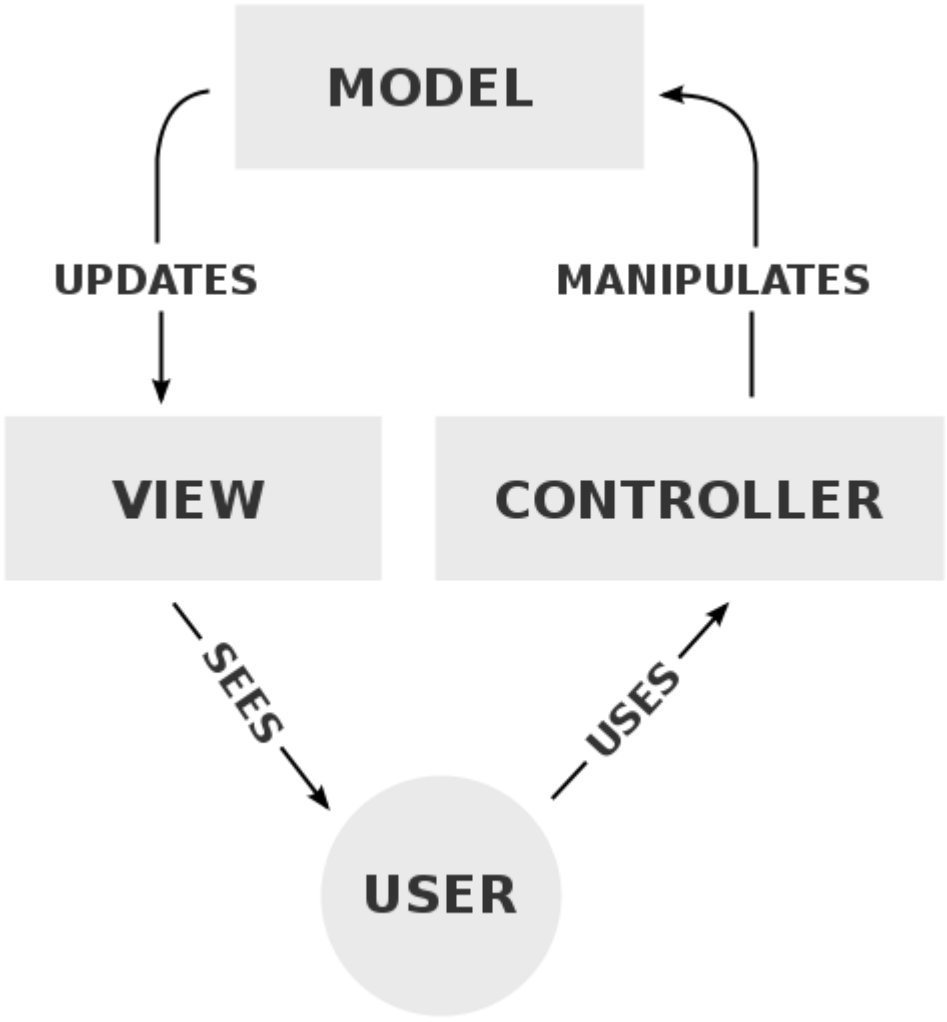


FIGURE 11. Typical MVC workflow

The shop's back end differs from the traditional example, in how it does not really use models that much. Nor does it directly update the state of the views (the modules that use AngularJS in the front end actually bind some data directly to a model). The MVC structure of the shop as a whole is better described by Figure 12 where the Model would be replaced by the eCommerce suite's API from where the shop retrieves its data (described in more detail in its own subchapter).

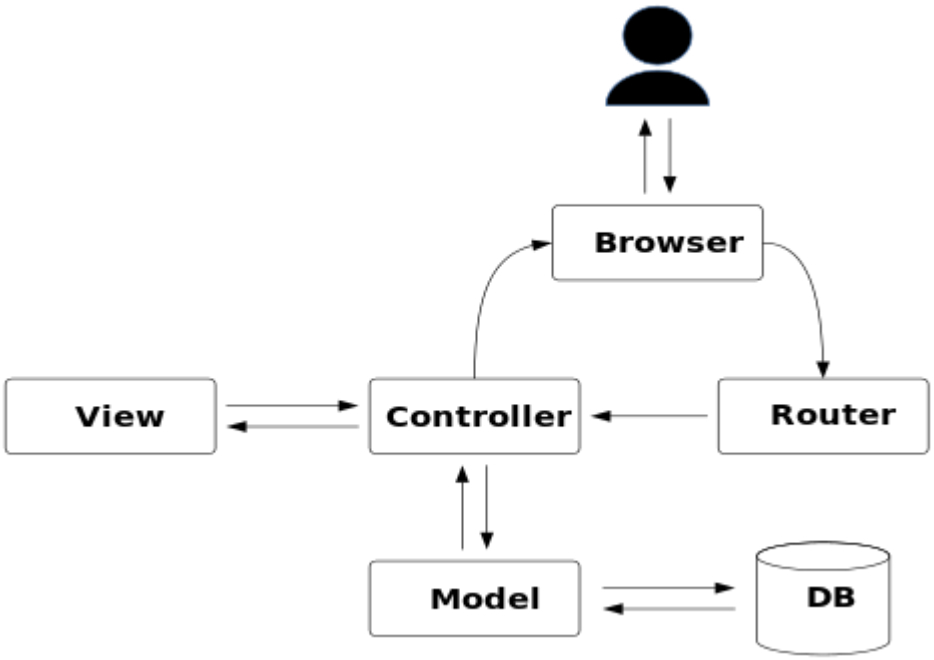


FIGURE 12. A typical collaboration of the MVC components (jmkim dot com, licensed under CC BY-SA 4.0)

4.2 Routing and responses

The application's HTTP routes, optional parameters, middleware and mappings are defined in the app/Http/routes.php (Otwell 2016d). Most of the shop's routes are mapped directly to some controller action. An example of HTPP GET routes with parameters mapped to controller methods can be seen in Figure 13 on the following page.

After the routing, the controllers inside the app/Http/Controllers' directory load some type of suitable response for the selected route. This could be, for instance, a Blade view as seen in Figure 14 on line 63, where the getPageById method in the PagesController controller (app/Http/Controllers/PagesController.php) returns a Blade view as the response content using Laravel's view helper method. It could also be a JSON response for AJAX calls made from an AngularJS controller after handling possible pre-requisites (such as retrieving, storing or parsing data required by the view in question). Figure 15 shows the addToCart method in the CartsController controller (app/Http/Controllers/CartsController.php)

returning a JSON reponse on line 48. This request could, for instance, originate from the addToCart function in the CategoryProductsController AngularJS controller (assets/js/modules/controllers/CategoryProductsController.js in the theme's directory under resources). This can be seen in Figure 16, where an HTTP POST request to the back end takes place.

Some pre-requisites, such as checks for authenticated users are handled by middleware assigned directly through certain routes in the routes file (Figure 17).

```
62  /**
63   * PAGES
64   */
65  Route::get('page/{id}-{slug?}', 'PagesController@getPageById');
66
67  /**
68   * PRODUCTS
69   */
70  // Show product
71  Route::get('product/{id}-{slug?}', '
        ProductsController@getProductById');
72
```

FIGURE 13. HTTP GET routes mapped to controller methods

```
46      /**
47       * Get page by id
48       *
49       * GET /page/{id}-{slug?}
50       *
51       * @param $id
52       * @param $slug Used for SEO, has no real purpose in method
         below
53       *
54       * @return Response
55       */
56      public function getPageById($id = null, $slug = false)
57      {
58          $page = Eshop::api_get('pages/id/' . $id);
59          $page = json_decode($page->getContent());
60
61          $page = $this->pageTransformer->transform($page->data);
62
63          return view($this->theme_path . 'pages/cmspage')->withPage(
                $page);
64      }
```

FIGURE 14. Returning a view in the PagesController class

```php
    /**
     * Add Product to Cart
     *
     * POST /a/add_to_cart
     *
     * @param  Request $request
     * @return JSON
     */
    public function addToCart(Request $request)
    {
        $product_id = (int) $request->product_id;
        $qty = 1;
        if ($request->has('qty')) {
            $qty = (int) $request->qty;
        }
        if (!Cart::add($product_id, $qty)) {
            return response()->json(['message' => 'error']);
        }
        return response()->json(['message' => 'added', 'alert' =>
            trans('messages.added_to_cart')]);
    }
```

FIGURE 15. Returning a JSON response in the CartsController class

```javascript
    // Add to cart
    $scope.addToCart = function(product_id, qty){
        $scope.add_in_progress[product_id] = true;

        $http.post('/a/add_to_cart', {
            product_id : product_id,
            qty : qty
        }).success(function(response){
            if (response.message === 'added') {
                alertMessage.set(response.alert, 'success');
            } else {
                alertMessage.set(generic.error, 'error');
            }
        }).error(function() {
            alertMessage.set(generic.error, 'error');
        }).finally(function(){
            $scope.add_in_progress[product_id] = false;
        });
    };
```

FIGURE 16. POST request in AngularJS controller

```
40  /**
41   * CUSTOMER DASHBOARD
42   */
43  Route::group(['middleware' => 'auth'], function () {
44      Route::get('dashboard', 'CustomerDashboardController@index');
45
46      Route::get('dashboard/orders', '
            CustomerDashboardController@orders');
47
48      Route::get('dashboard/customer', '
            CustomerDashboardController@editCustomer');
49      Route::post('dashboard/customer', '
            CustomerDashboardController@updateCustomer');
50
51      Route::get('dashboard/addresses', '
            CustomerDashboardController@addresses');
52      Route::get('dashboard/address', '
            CustomerDashboardController@createAddress');
53      Route::get('dashboard/address/{identifier}', '
            CustomerDashboardController@editAddress');
54      Route::post('dashboard/address', '
            CustomerDashboardController@storeAddress');
55      Route::put('dashboard/address', '
            CustomerDashboardController@updateAddress');
56      Route::delete('dashboard/address', '
            CustomerDashboardController@deleteAddress');
57
58      Route::get('dashboard/password', '
            CustomerDashboardController@editPassword');
59      Route::post('dashboard/password', '
            CustomerDashboardController@updatePassword');
60  });
```

FIGURE 17. Auth middleware assigned to route group

4.3 Modules

The modules are the core of the shop's custom functionality and features and a vital part of the applications information workflow. The term "modules" is in this context used as an umbrella term for all of the shop's custom service providers, facades as well as sub-views, which require an extensive amount of their "own" logic in the form of AngularJS controllers, services/factories and/or backend processing. The modules can be loosely divided into:

- service modules: A custom class along with a service provider and possibly a facade.
- template modules: Blade sub-views that require their own set of back end or front end logic in the form of Laravel or AngularJS controllers and services.

The template modules always rely on some back end controller or service module (otherwise, they would be referred to as “partials” instead). The service modules can be used by back end controllers or in a Blade template file through direct service injection.

4.3.1 Service modules

The service modules are custom services found under app/Http/Modules inside their own subdirectories and consist of a Laravel service provider, a custom class for the functionality (bound to the Laravel service container via the provider) and more often than not, a facade for convenient access.

Service providers extend Laravel’s abstract Illuminate\Support\ServiceProvider class and are used for registering bindings into Laravel’s service container, i.e. allowing, for instace, a custom class to be available for use throughout the application without manual instantiation (Otwell 2016f). A Laravel facade extends the Illumnate\Support\Facades\Facade class and serves as a “static proxy” to the classes in the service container, allowing access to objects from the container (Otwell 2016c). The service provider can be registered in config/app.php (Otwell 2016f).

4.3.2 Template modules

The template modules consist of one or more Blade views. For instance, the FeaturedCategories module is meant for displaying a specific set of “featured” categories on various pages or in various parts of a template, so it stands to reason that it may have several different views or variants depending on where it needs to be displayed. As opposed to the partials, the modules usually also have some logic in the form of an AngularJS controller, directive, service, factory or other code. If backend logic is needed, the modules can also have their own Laravel controllers and services. The Blade views and the front end assets (JavaScript, SASS) are located under resources/views/themes/<theme name>/assets (Figure 18) and divided as follows:

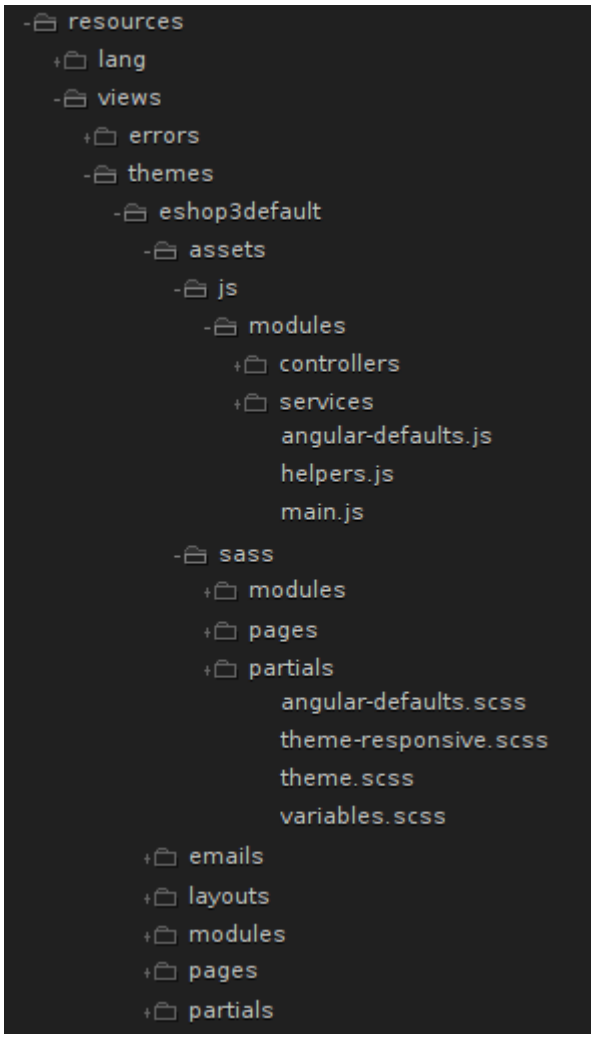


FIGURE 18. The theme's assets directory

- js/modules/controllers/<module name>Controller.js: AngularJS controller named after the module using it, e.g. FeaturedCategoriesController.js for the FeaturedCategories module. Besides the controller, the files may also contain AngularJS directives.
- js/modules/service/<module name>Factory.js or js/modules/service/<module name>Service.js: AngularJS service or factory named after the module using it, e.g. AlertMessagesFactory.js for the AlertMessages module.
- sass/modules/<module name>.scss: SASS stylesheet specific to the module, e.g. FeaturedCategories.scss
- modules/<module name>: This directory contains the Blade views for the module. Can contain multiple files, usually the default file is

named after the module, e.g. modules/QuickCart/QuickCart.blade.php for the QuickCart module's default view.

Should the module require its own back end Laravel controller for something like an AngularJS AJAX requests, it would be in app/Http/Controllers/Modules/<module name>/<module name>Controller.php. Some modules also have their own service module in app/Modules.

4.3.3 Information workflow and examples

This chapter contains examples of typically structured modules that are currently present and functional in the shop, as well their workflow explained through figures.

The FullSearch module, is a typical example of a template module. Consisting of a Blade view (Figure 19), an AngularJS controller (Figure 20), and its own SASS stylesheet (Figure 21), most of the template modules are built following this pattern. Notice the Blade view in Figure 19 defining the AngularJS controller for the module through the ng-controller attribute on line 2. Also, notice some of the brackets being prefixed with the @-sign, in order to be parsed by AngularJS rather than Blade. This module is designed to make AJAX calls from its AngularJS controller to perform simultaneous "live" searches for products, categories and pages related to a search query. Figure 22 shows the AngularJS controller watching for changes to the search query (line 43) and querying the back end for results through an XMLHttpRequest. The module does not have its own back end controller, but queries a more generic controller (not limited to any specific module), SearchController (app/Http/SearchController.php) for results (Figure 22). Figure 23 shows the module being loaded in a view using the @include directive on line 10.

```
<!-- Module FullSearch -->
<div class="module_fullsearch" ng-controller="FullSearchController"
>
    <div class="fullsearch">
        <h1>{{ trans('modules/FullSearch/strings.title') }}</h1>

        <input type="search" name="m_fullsearch_searchQuery" id="
        m_fullsearch_searchQuery" value="" placeholder="{{ trans('
            actions.search') }}..." class="form-control"
               ng-model="searchQuery"
               ng-init="searchQuery='{{ $search_query or '' }}'" />

        <div class="row">
            <!-- Products-->
            <div class="col-xs-12 col-sm-12 col-md-4">
                <h2>{{ trans('modules/FullSearch/strings.products'
                    ) }}</h2>
                <div ng-show="spinner_products" class="loader"></
                div>
                <span ng-show="!products">{{ trans('modules/
                    FullSearch/strings.no_results') }}</span>
                <ul class="media-list" data-ng-cloak>
                    <li ng-repeat="product in products" class="
                    media">
                        <div class="media-left">
                            <div>
                                <img class="media-object" ng-if="
                                product.images" ng-src="@{{ product
                                .images[0].thumb }}" alt="" />
                                <i class="fa fa-2x fa-angle-right"
                                ng-if="!product.images"></i>
                            </div>
                        </div>
                        <div class="media-body">
                            <h3 class="media-heading">
                                <a href="{{ URL::to('product') }}/
                                @{{ product.id }}-@{{ product.slug
                                }}">@{{ product.name }}</a>
                            </h3>
                        </div>
                    </li>
                </ul>
            </div>
```

FIGURE 19. The FullSearch module's Blade view

```
eshopApp.controller('FullSearchController', ['$scope', '$http',
    function($scope, $http) {

    $scope.searchQuery = null;

    $scope.search = function() {

        $scope.spinner_products = true;
        $scope.spinner_categories = true;
        $scope.spinner_pages = true;

        $http.post('/a/search/products', {
            searchQuery: $scope.searchQuery,
        }).success(function(response){
            if (response.data) {
                $scope.products = response.data;
            }
        }).finally(function(){
            $scope.spinner_products = false;
        });

        $http.post('/a/search/categories', {
            searchQuery: $scope.searchQuery,
        }).success(function(response){
            if (response.data) {
                $scope.categories = response.data;
            }
        }).finally(function(){
            $scope.spinner_categories = false;
        });

        $http.post('/a/search/pages', {
            searchQuery: $scope.searchQuery,
        }).success(function(response){
            if (response.data) {
                $scope.pages = response.data;
            }
        }).finally(function(){
            $scope.spinner_pages = false;
        });
    }

    // Init
    $scope.$watch('searchQuery', function() {
        $scope.search();
    });

}]);
```

FIGURE 20. The FullSearch module's AngularJS controller

```scss
@import "../variables.scss"; // SASS variables

.module_fullsearch {
    .fullsearch {

        ul.media-list  {
            li {
                div.media-left, div.media-right, div.media-body {
                    vertical-align: middle;
                }

                div.media-left {
                    text-align: center;

                    & > div {
                        width: 50px;

                        img {
                            width: 50px;
                            height: auto;
                        }
                    }
                }

                .media-heading {
                    margin-top: 5px;

                    a {
                        color: $black;
                        &:hover {
                            text-decoration: underline;
                        }
                    }
                }

            }

        }
    }
}
```

FIGURE 21. The FullSearch module's SASS stylesheet

```
11  class SearchController extends Controller
12  {
13      protected $productTransformer;
14      protected $categoryTransformer;
15      protected $pageTransformer;
16
17      protected $search = '';
18      protected $param_string;
19      protected $limit = 50;
20      protected $page = 1;
21
22      protected $categories_search_level = 2;
23
24      public function __construct(Request $request,
            ProductTransformer $productTransformer,
            CategoryTransformer $categoryTransformer, PageTransformer
            $pageTransformer)
25      { ...
38      }
39
40      /**
41       * Load the search result page
42       *
43       * GET /search
44       *
45       * @return Response
46       */
47      public function index()
48      {
49          return view($this->theme_path.'pages/search/results')->
                with(['search_query' => $this->search]);
50      }
51
52      /**
53       * Search for Products
54       *
55       * POST /a/search/products
56       *
57       * @return JSON
58       */
59      public function searchProducts()
60      { ...
67      }
68
69      /**
70       * Search for Categories
71       *
72       * POST /a/search/categories
73       *
74       * @return JSON
75       */
76      public function searchCategories()
```

FIGURE 22. Fraction of the SearchController class

```
1  @extends($theme_path.'.layouts.default')
2
3  @section('title')
4      {{  trans('pages.search') }}
5  @endsection
6
7  @section('content')
8
9      {{-- FullSearch Module --}}
10     @include($theme_path.'/modules/FullSearch/FullSearch', compact(
           $search_query))
11
12 @endsection
```

FIGURE 23. Loading the FullSeach module

Another module worth mentioning is the ContactForms module, for generating and processing various contact forms. This module qualifies as both a template module and a service module, since it requires both in order to function. The module does not have an AngularJS controller, since no AJAX or other interactivity that would require the use of JavaScript is currently present, but rather fetches the data it needs directly from the ContactForms service (Figure 24) injected into the module's view (Figure 25). The injection of the ContactForms service (app/modules/ContactForms/ContactForms.php) into the ContactForms view (modules/ContactForms/ContactForms.blade.php inside the theme's directory) through the @inject directive can be seen in Figure 25 on line 6.

```php
<?php
namespace App\Modules\ContactForms;

class ContactForms
{

    /**
     * Name of the ContactForms config file in app/config
     * @var string
     */
    private $config = 'contactforms';

    /**
     * Check if an active contact form exists for the given handle
     *
     * @param  string $form_handle
     * @return boolean
     */
    public function exists($form_handle)
    {
        return in_array($form_handle, config($this->config . '.
            active_forms'));
    }

    /**
     * Load all form settings from config
     *
     * @param  string $form_handle
     * @return array
     */
    public function get($form_handle)
    {
        return config($this->config . '.form.' . $form_handle);
    }
```

FIGURE 24. Fraction of the ContactForms service

```
@if(!isset($form_handle))
    <strong>THIS MODULE REQUIRES A CONTACT FORM HANDLE TO BE LOADED IN ADVANCED OR INJECTED INTO VIEW ON LOAD</strong>
@else

    {{-- Inject the ContactForms service --}}
    @inject('contactforms', 'App\Modules\ContactForms\ContactForms')
    @if (!$contactforms->exists($form_handle))
        <strong>NO ACTIVE CONTACT FORM EXISTS FOR SELECTED HANDLE</strong>
    @endif

    <!-- Module ContactForms -->
    <div class="module_contactforms">

        @if ($errors->has())
            <!-- Error message -->
            <div class="alert alert-danger" role="alert">
                <strong>{{ trans('messages.form_has_errors') }}</strong>
            </div>
        @endif

        <form action="{{ URL::to('m/contactforms') . '/' . $form_handle }}" method="POST" class="contactforms" name="contactform">...

        </form>
    </div>
    <!-- /Module ContactForms -->

@endif
```

FIGURE 25. The ContactForms module's view

The ContactForms service also needs a service provider and a facade in order to be registered by the application. Figure 26 below shows the service provider of the ContactForms module registering the ContactForms class into Laravel's service container. The facade can be seen in Figure 27.

```php
<?php
namespace App\Modules\ContactForms;

use Illuminate\Support\ServiceProvider;

class ContactFormsServiceProvider extends ServiceProvider
{
    /**
     * Register the service provider
     *
     * @return void
     */
    public function register()
    {
        $this->app->bind('contactforms', function () {
            return new ContactForms();
        });
    }
}
```

FIGURE 26. The service provider

```php
<?php
namespace App\Modules\ContactForms;

use Illuminate\Support\Facades\Facade;

class ContactFormsFacade extends Facade
{
    /**
     * Name of IoC binding is...
     *
     * @return string
     */
    public static function getFacadeAccessor()
    {
        return 'contactforms';
    }

}
```

FIGURE 27. The facade

Since the back end logic for processing the forms needs to be very specific in order to fit the module, it also features its own, dedicated back end controller for processing the POST data from the forms. Figure 28 shows a fraction of the ContactForms module's controller (app/Http/Modules/ContactForms/ContactFormsController.php). Notice the controller calling the ContactForm service's get method through the facade (line 34) mentioned earlier.

```php
<?php
namespace App\Http\Controllers\Modules\ContactForms;

use App\Http\Controllers\Controller;
use ContactForms;
use Illuminate\Http\Request;
use Mail;
use Validator;

class ContactFormsController extends Controller
{
    public function __construct()
    {
        parent::__construct();
    }

    /**
     * Handle contact form
     *
     * POST /contact/{form_handle}
     *
     * @param  Request $request form data
     * @param  string  $form_handle
     * @return Response
     */
    public function index(Request $request, $form_handle = false)
    {
        // Not activated in contactforms config
        if (!$form_handle || !ContactForms::exists($form_handle)) {
            return back();
        }

        // Load form settings from config
        $form = ContactForms::get($form_handle);

        // Parse fields
        $data = [];
        foreach ($form['fields_rules'] as $field => $rule) {
            if ($request->has($field)) {
                $data[$field] = $request->input($field);
            }
        }

        // Validate
        $validator = Validator::make($data, $form['fields_rules']);
        if ($validator->fails()) {
            return back()->withErrors($validator)->withInput();
        }
```

FIGURE 28. Fraction of the ContactForms module's controller

Yet another module that differs from the ones mentioned above is the Eshop module. This service module fetches any needed data from the eCommerce platform's API, e.g. products, categories, pages etc. This is an example of a module that is not directly related to or used by any of the template modules. It is, however, crucial to several back end controllers and services. For instance, the SearchController mentioned earlier utilizes

the Eshop module (through the Eshop module's facade) to fetch data from the API. The logic of this module is described more thoroughly later on in the text.

4.4 Retriving data

4.4.1 Sources of data

The shop needs to retrieve and manipulate a vast number of data regarding products, customers, orders and other things expected of the front office for an eCommerce platform. The shop is not directly connected to a relational database (i.e. does not directly execute SQL queries via its back end), but fetches and stores data by communicating with the eCommerce platform's API (which, in turn, is connected to a relational database, although that's beyond the scope of this report) using HTTP requests. Sessions and cookies are also used by the shop to help keep track of some of the changing data, such as the shopping cart and currently logged in customers, or simply to remember some choices made by a visitor. However, most of the dynamic data that the shop requires is constantly retrived and stored back and forth through the API mentioned above, even the authenticatable users and the actual contents of the shopping cart (the cookie merely acts as an identifier).

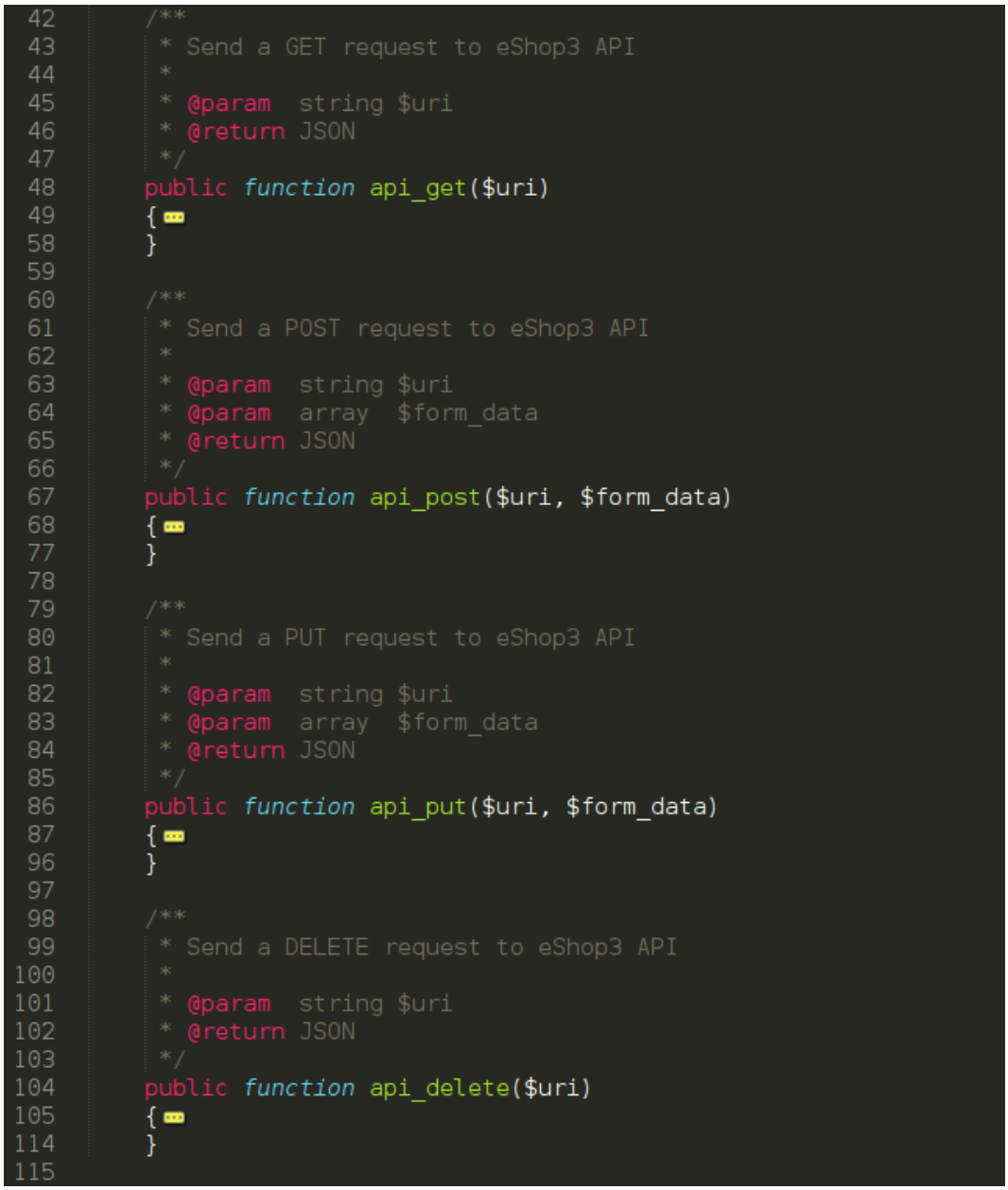
4.4.2 Communicating with the API

The shop transfers data to and from the eCommerce suite's API (after this referred to simply as "the API") via an HTTP request using the Eshop service module (under app/Modules/Eshop). This module makes use of the Guzzle client for its requests. Guzzle is a PHP HTTP client that provides a simple interface for asynchronous and synronous requests (Dowling 2016). It can be installed using Composer (Dowling 2016). The Guzzle client is injected through the module's service provider (app/Modules/Eshop3/EshopServiceProvider.php), seen in Figure 29 below.

The EshopAPI service (app/Modules/Eshop3/EshopAPI.php), seen in Figure 30, contains readily made methods for HTTP GET, POST, PUT and DELETE requests, which are used in a majority of the back end controllers throughout the application via the module's facade. Figure 31 shows the ProductsController (app/Http/Controllers/ProductController.php) controller fetching API data by calling the service's api_get method via the Eshop facade. The call to the service through the facade can be seen on line 37.

```
1  <?php
2  namespace App\Modules\Eshop3;
3
4  use Illuminate\Support\ServiceProvider;
5  use Config;
6
7  class EshopServiceProvider extends ServiceProvider {
8
9      /**
10      * Register the service provider
11      *
12      * @return void
13      */
14     public function register()
15     {
16         header("Access-Control-Allow-Origin: ".config('eshopAPI.
               domain'));
17
18         $this->app->bind('eshop', function(){
19             $client = new \GuzzleHttp\Client([
20                 'base_uri' => config('eshopAPI.api_url'),
21                 []
22             ]);
23             return new EshopAPI($client);
24         });
25     }
26 }
```

FIGURE 29. The Eshop service provider



```
42      /**
43       * Send a GET request to eShop3 API
44       *
45       * @param  string $uri
46       * @return JSON
47       */
48      public function api_get($uri)
49      { ...
58      }
59
60      /**
61       * Send a POST request to eShop3 API
62       *
63       * @param  string $uri
64       * @param  array  $form_data
65       * @return JSON
66       */
67      public function api_post($uri, $form_data)
68      { ...
77      }
78
79      /**
80       * Send a PUT request to eShop3 API
81       *
82       * @param  string $uri
83       * @param  array  $form_data
84       * @return JSON
85       */
86      public function api_put($uri, $form_data)
87      { ...
96      }
97
98      /**
99       * Send a DELETE request to eShop3 API
100      *
101      * @param  string $uri
102      * @return JSON
103      */
104     public function api_delete($uri)
105     { ...
114     }
115
```

FIGURE 30. Fraction of the EshopAPI service

```
24      /**
25       * Get product by id and display product info
26       *
27       * GET /product/{id}-{slug}
28       *
29       * @param int $id
30       * @param string $slug Used for SEO, has no real purpose in
         method below
31       *
32       * @return response
33       */
34      public function getProductById($id = false, $slug = false)
35      {
36          // Get product from API
37          $product = Eshop::api_get('products/id/' . $id);
38          $product = json_decode($product->getContent());
39
40          // Transform fields
41          $product = $this->productTransformer->transform($product->
                data);
42
43          return view($this->theme_path . 'pages/product')->
                withProduct($product);
44      }
```

FIGURE 31. Controller method fetching API data though the Ehop facade

4.4.3 Securing the communication

To secure the communication between the shop and the API, the module uses Hash-based Message Authentication Code (HMAC) implemented through PHP's hash_hmac function.

As the articles "Create a HMAC-SHA authentication implementation for PHP" by P. Brown (2016) and "API Authentication: HMAC with Public/Private Hashes" by C. Cornutt (2016) explain, the idea behind HMAC is that both participants have access to a set of so-called public and private keys. The public key, which could be some sort of random character string, is used to identify the shop, while the private key is used when generating some sort of hash or Message Authentication Code (MAC). The shop and the API both know the values of the public and private keys. The shop then generates a hash where the private key is included. The hash is passed along as a header with the public key in every request to the API. The API, which recognizes the shop through the public key, then attempts to generate the same hash using the private key. If the hashes match when compared, the API knows that both must have

been using the same private key. The private key itself is never passed along, so it will not be compromised even if the generated hash is intercepted. The hash itself could be used though, unless it contains some sort of changing value. Figure 32 illustrates the process of a hash (MAC) being generated on both sides using the private key (K), and verified. In this example, the hash is generated from the message itself along with the private key (K), although the hash generated using the private key could as well be based on some other token as long as both the sender and the receiver know what the hashing process is based upon.

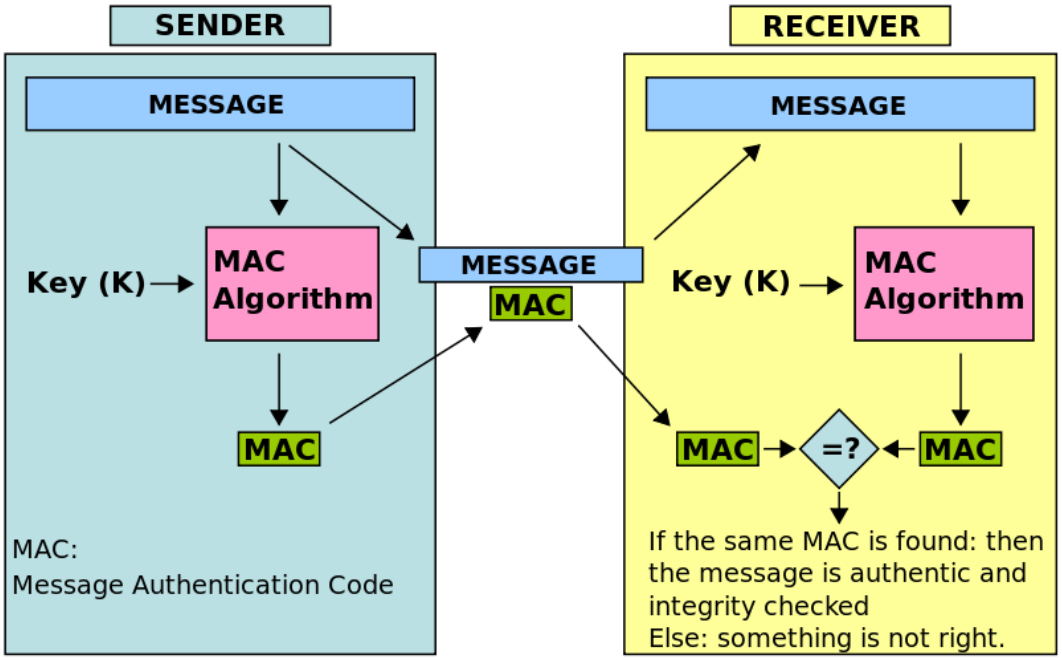


FIGURE 32: Example of HMAC-based communication

4.5 Transforming data

4.5.1 The need for transformation

Since the data from the API may contain various fields and data that the shop may or may not want to make directly available to the front end, the data needs to be manipulated or “transformed” by the back end before use. This is especially important in the case of AJAX requests, since the data that is returned is directly accessible through the browser (though it may or may not actually be clearly visible to the naked eye in the user

interface). In most cases, this simply means defining which properties of the object obtained from the API the array used for final output to the front end should contain. Some of the transformers also contain functionality beyond simple definitions, such as various calculations and permission checks. that the shop needs for a certain entity, be it by the back end or the front end. Many types of data, such as information regarding the shopping cart or product data is needed by several controllers in serveral different places of the shop. Thus, having a centralized way of transforming the data is vital for re-usability and for standardizing the output. This means not having to calculate and/or define the same things over and over again every time some type of data is fetched from the API.

4.5.2 Transformers

The data tranformers are classes located within the app/Transformers directory and they all extend from the same abstract Transformer parent class (app/Transformers/Transformer.php). The abstract transformer class, seen in Figure 33, contains readily made methods for transforming objects and arrays, either as single items or as a collection of items. Notice the abstract method, transform, defined on line 63. This method contains no logic of its own, but must be defined by all the other transformers that extend the Transformer class. It is also needed by the Transformer's transformCollection and transformArrayCollection methods, since both utilize the transform method for transforming data.

The child classes are only required to contain one single method, transform (as defined by an abstract method with the same name in the Transform parent class), which handles the transformation of a certain entity, e.g. customer, product or shopping cart. The transformers are usually injected into the back end controllers on a per-required basis through the controller's constructor. Figure 34 illustrates the ProductTransformer transformer (app/Transformers/ProductTransformer.php) being injected into the ProductsController controller (app/Http/Controllers/ProductsController.php)

through the class constructor on line 18. Notice that the full namespace of the transformer is actually App\Transformers\ProductsTransformer as defined by the use keyword before the class on line 8. The transformer can then be used to transform a collection of data or a single object, both of which are illustrated by Figures 35 and 36. Figure 35 illustrates the CategoryProducts module-specific controller (app/Http/Controllers/Modules/CategoryProducts/CategoryProductsController.php) using the ProductTransformer transformer (app/Transformers/ProductTransformer.php) to transform a collection of products (objects) from the API (line 56) before returning it as a JSON response to the front end (line 57). Figure 36 illustrates the ProductsController controller (app/Http/Controllers/ProductsController.php) using the ProductTransformer transformer to transform a single product / object (line 41) before returning the reponse, i.e. loading the product information view and passing along the data (43).

```php
<?php
namespace App\Transformers;

abstract class Transformer
{

    /**
     * Transform a collection of API data before final output to
     front end
     *
     * @param JSON $json_items
     * @return array $transformed
     */

    public function transformCollection($json_items)
    {
        $items = json_decode($json_items->getContent());

        $transformed = $this->setPagination($items);

        foreach ($items->data as $item) {...
        }

        return $transformed;
    }

    /**
     * Transform a collection of array data before final output to
     front end.
     * This is useful for quickly transforming a collection of
     related items, such as Cart Products
     *
     * @param  array $array_items
     * @return array $transformed
     */
    public function transformArrayCollection($array_items)
    {
        foreach ($array_items as $item) {
            $transformed['data'][] = $this->transform($item);
        }
        return $transformed;
    }

    /**...
    public function setPagination($items)
    {...
    }

    abstract public function transform($item);
}
```

FIGURE 33. The abstract Transformer class

```php
<?php
namespace App\Http\Controllers;

use Auth;
use App\Helpers\Helper;
use App\Http\Controllers\Controller;
use App\Modules\Eshop3\Eshop;
use App\Transformers\ProductTransformer;
use Illuminate\Http\Request;

class ProductsController extends Controller
{
    /**
     * @var Transformers\ProductTransformer
     */
    protected $productTransformer;

    public function __construct(ProductTransformer $
        productTransformer)
    {
        parent::__construct();
        $this->productTransformer = $productTransformer;
    }
```

FIGURE 34. Injecting the transformer through the constructor

```php
    /**
     * Get products by category from API and return to module
     *
     * GET /m/products/category/{category_id}
     *
     * @param  int $category_id
     * @return JSON
     */
    public function getProductsByCategory($category_id = false)
    {
        $products = Eshop::api_get('products/categories/id/' . $
            category_id . $this->additional_param_string);
        $output = $this->productTransformer->transformCollection($
            products);
        return response()->json($output);
    }
```

FIGURE 35. Transforming a collection of items

```
24      /**
25       * Get product by id and display product info
26       *
27       * GET /product/{id}-{slug}
28       *
29       * @param int $id
30       * @param string $slug Used for SEO, has no real purpose in
         method below
31       *
32       * @return response
33       */
34      public function getProductById($id = false, $slug = false)
35      {
36          // Get product from API
37          $product = Eshop::api_get('products/id/' . $id);
38          $product = json_decode($product->getContent());
39
40          // Transform fields
41          $product = $this->productTransformer->transform($product->data
                );
42
43          return view($this->theme_path . 'pages/product')->withProduct(
                $product);
44      }
```

FIGURE 36. Transforming a single item

4.5.3 Examples and variations

The CustomerTransformer transformer (app/Transformers/CustomerTransformer.php) illustrated in Figure 37 is a simple example of a typical transformer class. This transformer only defines and type-casts various properties of a customer (object) obtained from the API and has no other methods beyond the obligatory transform method. Notice that the customer object may contain more properties than the ones that are returned by its transformer.

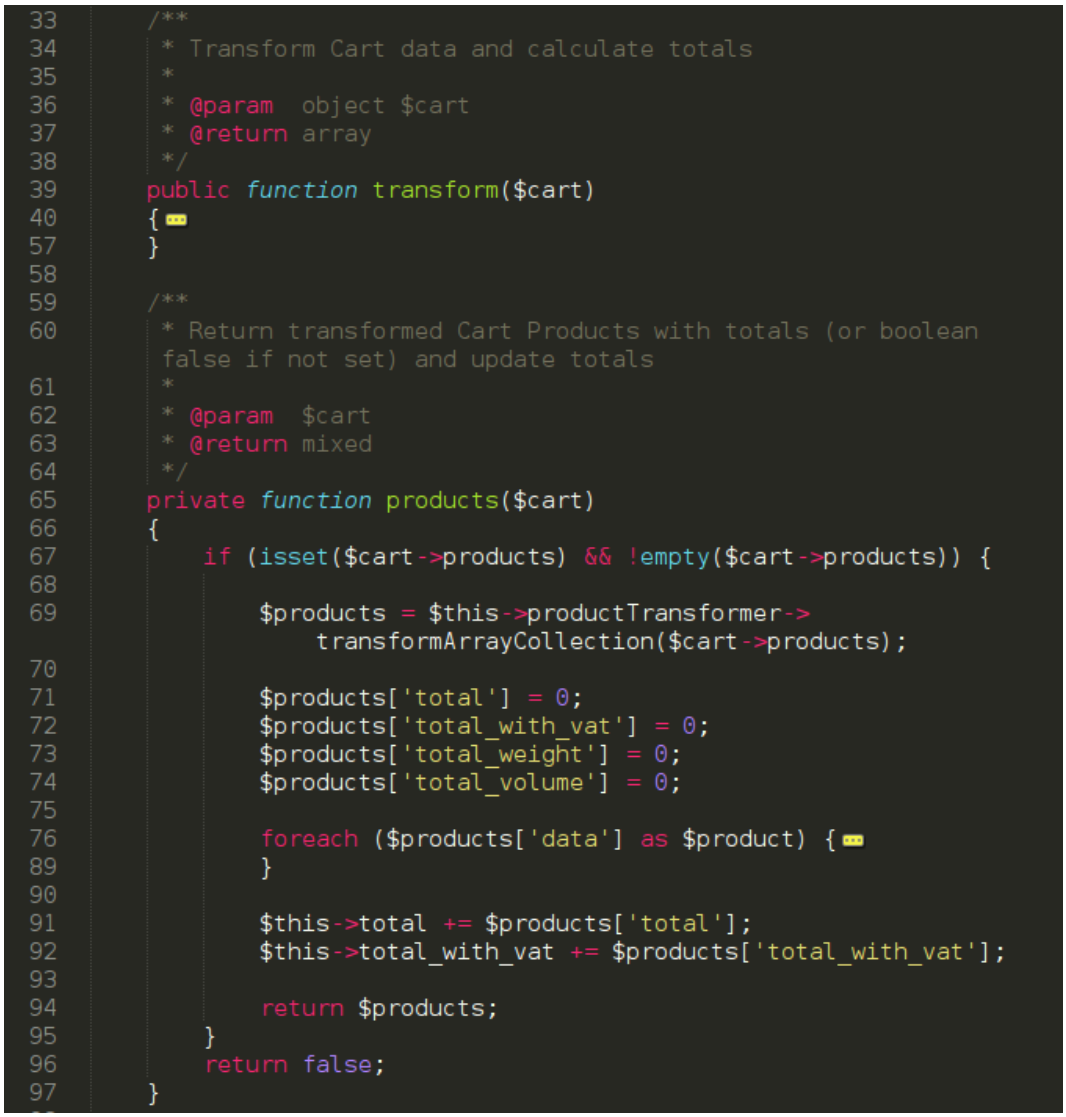
```php
<?php
namespace App\Transformers;

class CustomerTransformer extends Transformer {

    /**
     * Define/transform customer fields for frontend
     *
     * @param  object $customer
     * @return array
     */
    public function transform($customer)
    {
        return [
            'registered' => (bool) $customer->registered,
            'firstname' => $customer->first_name,
            'lastname' => $customer->last_name,
            'company' => $customer->company,
            'vatnumber' => $customer->business_id,
            'email' => $customer->email,
            'phone' => $customer->phone,
        ];
    }
}
```

FIGURE 37. The CustomerTransformer transformer

The CartTransformer transformer (app/Transformers/CartTransformer.php) illustrated in Figure 38 serves well as a more advanced example. This transformer contains multiple methods and has several tasks vital to the functionality of the shop beyond the scope of simply determining which object properties are needed. For instance, it calculates the total sum, weight and volume of the shopping cart and related products. Notice that it also relies on other transformers to work, such as the ProductTransformer transformer for transforming the products (line 69).



```
33      /**
34       * Transform Cart data and calculate totals
35       *
36       * @param  object $cart
37       * @return array
38       */
39      public function transform($cart)
40      {...
57      }
58
59      /**
60       * Return transformed Cart Products with totals (or boolean
         false if not set) and update totals
61       *
62       * @param  $cart
63       * @return mixed
64       */
65      private function products($cart)
66      {
67          if (isset($cart->products) && !empty($cart->products)) {
68
69              $products = $this->productTransformer->
                    transformArrayCollection($cart->products);
70
71              $products['total'] = 0;
72              $products['total_with_vat'] = 0;
73              $products['total_weight'] = 0;
74              $products['total_volume'] = 0;
75
76              foreach ($products['data'] as $product) {...
89              }
90
91              $this->total += $products['total'];
92              $this->total_with_vat += $products['total_with_vat'];
93
94              return $products;
95          }
96          return false;
97      }
```

FIGURE 38. Fraction of the CartTransformer transformer

SUMMARY

The project aimed at making an easily maintainable and expandable webshop as part of a larger eCommerce suite, using well-known MVC frameworks. The main challenges involved getting the different parts of the eCommerce suite working together and designing the overall structure and interaction between the webshop's various frameworks, as well as the interaction between the back and the front ends. Furthermore, the hierarchial structure and information flow would require a well thought-out structure early on in order to be open for changes, customization and expansion later on.

The current structure of the application allows both the themes and business logic to be customized in order to meet current and future needs presented by the customer/end-user. Themes and custom logic can be modified or switched completely. The shop can also be expanded with new features by adding various types of modules, should a need for completely new features arise. As future implementations and custom shop's are made and published, the total number of readily-made features, modules and themes to select from is naturally expected to grow. In the long run, this could be expected to speed up development and publishing as developers will only need to make minor improvements and modifications to existing modules and themes, rather than needing to actually develop completely new functionality.

At the time of writing, the shop is undergoing customer-specific customizations to its themes and modules in order to be deployed for the commissioning company's end-users. The first site is expected to go online during spring 2016.

SOURCES

Adermann, N. & Boggiano, J. 2016. Composer – Introduction [referenced 20 Feb 2016].
Available on: https://getcomposer.org/doc/00-intro.md

Bean, M. 2015. Laravel 5 Essentials.

Bower 2016. Bower [referenced 1 March 2016].
Available on: http://bower.io/

Branas, R. 2014. AngularJS Essentials. Packt Publishing Ltd.

Brown. P. 2014. Create a HMAC-SHA authentication implementation for PHP [referenced 12 March 2016].
Available on: http://culttt.com/2014/05/21/create-hmac-sha-authentication-implementation-php/

Cornutt, C. 2013. API Authentication: HMAC with Public/Private Hashes [referenced 1 March 2016].
Available on: https://websec.io/2013/02/14/API-Authentication-Public-Private-Hashes.html

DevNet Oy. 2016. DevNet Oy – DevNet lyhyesti [referenced 1 Feb 2016].
Available on: http://www.devnet.fi/yritys

Dowling, M. 2015. Guzzle, PHP HTTP client – Guzzle Documentation [referenced 1 March 2016].
Available on: http://docs.guzzlephp.org/en/latest/index.html

Dowling, M. 2015. Overview – Guzzle Documentation [referenced 1 March 2016].
Available on: http://docs.guzzlephp.org/en/latest/overview.html#installation

Freeman, A. 2014. Pro AngularJS. Apress.

Green, B. & Seshadri, S. 2013. AngularJS. O'Reilly Media, Inc.

Maynard, T. 2015. Getting Started with Gulp. Packt Publishing Ltd.

Otwell, T. 2016a. Application structure – Laravel – The PHP Framework For Web Artisans [referenced 1 March 2016].
Available on: https://laravel.com/docs/5.1/structure

Otwell, T. 2016b. Blade Templates – Laravel – The PHP Framework For Web Artisans [referenced 1 March 2016].
Available on: https://laravel.com/docs/5.1/blade

Otwell, T. 2016c. Facades – Laravel – The PHP Framework For Web Artisans [referenced 1 March 2016].
Available on: https://laravel.com/docs/5.1/facades

Otwell, T. 2016d. HTTP Routing – Laravel – The PHP Framework For Web Artisans [referenced 1 March 2016].
Available on: https://laravel.com/docs/5.1/routing

Otwell, T. 2016e. Laravel Elixir – Laravel – The PHP Framework For Web Artisans [referenced 1 March 2016].
Available on: https://laravel.com/docs/5.1/elixir

Otwell, T. 2016f. Service Providers – Laravel – The PHP Framework For Web Artisans [referenced 1 March 2016].
Available on: https://laravel.com/docs/5.1/providers

Principe, M. & Yoon, D. 2015. Proceedings of the International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government (EEE) – A WEB APPLICATION USING MVC FRAMEWORKS.

Q-Success. 2016. Usage Statistics and Market Share of JavaScript Libraries for Websites, March 2016 [referenced 30 Feb 2016].
Available on:
http://w3techs.com/technologies/overview/javascript_library/all

Spurlock, J. 2013. Bootstrap. O'Reilly Media, Inc.

Surguy M. 2013. Maxoffsky - History of Laravel PHP framework, Eloquence emerging [referenced 20 Feb 2016].
Available on: http://maxoffsky.com/code-blog/history-of-laravel-php-framework-eloquence-emerging

Teixeira, P. 2012. Professional Node. js: Building Javascript based scalable software. John Wiley & Sons.

The jQuery Foundation. 2016. jQuery [referenced 20 Feb 2016].
Available on: http://jquery.com